Tackling the Time-Defence: An Instruction Count Based Micro-architectural Side-Channel Attack on Block Ciphers

Manaar Alam(✉), Sarani Bhattacharya, and Debdeep Mukhopadhyay

Indian Institute of Technology, Kharagpur, Kharagpur, India
alam.manaar@iitkgp.ac.in,
{sarani.bhattacharya,debdeep}@cse.iitkgp.ernet.in

Abstract. Hardware Performance Counters (HPCs) are present in most modern processors and provide an interface to user-level processes to monitor their processor performance in terms of the number of micro architectural events, executed during a process execution. In this paper, we analyze the leakage from these HPC events and present a new micro-architectural side-channel attack which observes number of instruction counts during the execution of an encryption algorithm as side-channel information to recover the secret key. This paperfirst demonstrates the fact that the *instruction* counts can act as a side-channel and then describes the *Instruction Profiling Attack* (IPA) methodology with the help of two block ciphers, namely AES and Clefia, on Intel and AMD processors. We follow the principles of profiled instruction attacks and show that the proposed attack is more potent than the well-known cache timing attacks in literature. We also perform experiments on ciphers implemented with popular time fuzzing schemes to subvert timing attacks. Our results show that while the countermeasure successfully stops leakages through the timing channels, it is vulnerable to the Instruction Profiling Attack. We validate our claims by detailed experiments on contemporary Intel and AMD platforms to demonstrate that seemingly benign instruction counts can serve as side-channels even for block cipher implementations which are hardened against timing attacks.

Keywords: Micro-architectural side-channel attack · Hardware performance counters · Cache-timing attack · Block-cipher

1 Introduction

The state of a computing environment gets affected by the processes executing on it. Modern cryptosystems are vulnerable against growing threats in the form of information leakages about the secret key through side-channels like power, radiation, timing, etc. One category of such threats uses the information leakages created because of the variations (in say timing) caused due to the presence of a cache memory in processors. The cause of the variation is that the access time for a data present in the cache is much lower than the data not present in it, as the processor first looks into the cache memory before processing with a data. This disparity in the access time is the fundamental notion of all cache-attacks. Block ciphers like Data Encryption Standard (DES), Advanced Encryption Standard (AES), Clefia, Blowfish, etc. are vulnerable to cache-attacks as they require key dependent table lookup for their encryption operations.

Two important classes of cache-attacks are - cache trace attacks and cache timing attacks. For a cache trace attack, an adversary needs to profile the cache access patterns, in terms of cache-hits and cache-misses during the encryption operation. Cache timing attacks, on the other hand, only require the information regarding overall execution time of the encryption process, thus making it more threatening than the other form of cache-attacks. An adversary can easily capture the timing information over a network, without the need of any sophisticated measuring instrumentation, thereby, creating a chance of possible remote attacks. It can be pointed here that these threats have been shown to be pertinent even in a remote server running an encryption algorithm [2,4].

There have been some seminal works of cache timing attack on block ciphers, both with large tables like AES and small tables like Clefia. Bernstein [4] demonstrated that statistical correlation between the profiles of the execution time of AES for a known key and an unknown key could be used to extract the secret key bytes. In the same way Rebeiro et al. [20,21] described that the timing profiles for the Clefia encryption for both the known and unknown key could be used to obtain the round keys and thereby trivially determine the secret key.

Numerous works have been done to counter the risks of cache timing attacks [8,11,16,19]. A notable work to prevent these attacks is described in [11] by Martin et al. They presented a general mitigation strategy to limit the fidelity of fine-grained time-keeping, thereby making it difficult for the adversary to distinguish between different time-stamps. The authors mainly focused on the RDTSC (Read Time Stamp Counter) instructions, which returns the current value of TSC (Time Stamp Counter) register, to design their countermeasures and eliminated the possibility of information leakages through other micro-architectural events. But, on the contrary to their claim, the advent of *perf_event* system call and performance monitoring tool PAPI [18] allows an adversary to monitor any micro-architectural event of a system with user privilege and with higher granularity. Also, the work reported in [6] has shown the possibility of timing attack in spite of the presence of time obfuscation. Most modern processors contain hardware performance counters which count the total occurrences of different micro-architectural events. The PAPI tool gives an upper hand to an adversary to mount an attack in the presence of the defense by time fuzzing technique.

A micro-architectural attack has been proposed based on Instruction cache (I-cache) [1] for public key cryptographic implementation, however, the hardware event *instruction* count has not been quite explored in the case of block ciphers. A possible reason could be as block ciphers do not have key-dependent conditional branches like public key ciphers, the number of instructions does not intuitively leak the secret key. In this paper, we look into this issue of exploring whether the micro-architectural event, instruction-count, can be utilized to reveal secret keys of block ciphers. We propose an Instruction Profiling Attack (IPA), which thus exploits this not-so-researched side-channel, i.e., instruction counts, and determines secret keys from block cipher executions faster than traditional cache timing attacks. A related fall-out of this attack is that, block cipher implementations which are time-resistant by fuzzing time stamp counters are still vulnerable against the proposed IPA.

The attack methodology is validated on two different types of block ciphers, namely AES and Clefia on two separate processors, Intel Core i5 and AMD A10.

Main Idea and Motivation

The cache timing attacks work on the intuition of non-uniform cache memory accesses due to cache hits and cache misses. These memory operations are nothing but simple load- store instructions spawned by the CPU. In the presence of timing obfuscation defense mechanism, an adversary is unable to analyze the timing differences and gain secret information but can monitor the number of instructions executed in the system. The total number of load-store instructions executed during the encryption process will vary based on the secret key, as the cache access pattern will be different for different secret keys. A cache-miss operation will result in an extra load instruction and this disparity in the executed instructions is the main idea behind this work. The motivation is to use the instruction count event as an information leakage source and mount an attack.

Most of the modern encryption implementations use time obfuscation technique, which can be implemented easily with little performance overhead, to mitigate the threats of cache-timing attacks. Distributions, like OSX, Ubuntu use deliberate time delay after entering the wrong password to invalidate the timing analysis done by an adversary [24]. The present paper strives to evaluate the security of such apparently secured systems against timing attacks in the face of the newly proposed threat.

Our Contribution

We have investigated the presence of information leakage of an encryption process using the hardware event *instruction* count. The prime contributions that we made through this work are:

– We have proposed a new side-channel attack, namely Instruction Profiling Attack (IPA), tailored for block ciphers using HPC event: *instruction*. We have also demonstrated the effectiveness of IPA by successfully retrieving the secret key bits.
– We have evaluated our proposed attack method on both Intel and AMD platforms. The time complexity of a successful IPA is shown to be much lesser than a successful cache-timing attack.

– Additionally, in this paper, we show that the success rate of IPA is not affected by time-obfuscation countermeasures like timewarp, which can successfully thwart attacks based on timing channels.

The rest of the paper is organized as follows: the next section presents an overview of the necessary preliminaries related to this work. Section 3 analyzes a new form of side-channel leakage by analyzing the security of block ciphers. Section 4 discusses our attack methodology with the help of a case study on AES block cipher. Section 5 demonstrates all the experimental results, and Sect. 6 discusses the practicality of the proposed attack methodology in different environments. Finally, Sect. 7 presents the conclusion of this work.

2 Preliminaries

In this section, we first discuss the basic operations of two block ciphers, namely AES and Clefia. Next, we describe a time obfuscating countermeasure to thwart cache-timing attacks. Then we present a brief overview of hardware performance counters which are instrumental to the proposed attack. We follow for evaluating the proposed attack methodology.

2.1 AES Block Cipher

AES [7] is a 10 round cipher which takes a 16 byte secret key $K = (k_0, k_1, \cdots, k_{15})$ and an input of 16 byte plain text $P = (p_0, p_1, \cdots, p_{15})$. Its implementation in software, based on Barreto's code, is widely recognized [3]. The first 9 rounds of the algorithm uses four 1 KB lookup tables T_0, T_1, T_2, and T_3 and then an additional look up table T_4 for the final round. Though the use of lookup tables optimizes the performance of the algorithm, the size of the lookup table is large. The following equation shows the structure of the cipher in each round, encapsulating the four basic operations of AES, namely the *SubBytes, ShiftRows, MixColumns* and *AddRoundKey*: For each round r, $(1 \leq r \leq 9)$ the input is the state S^r comprising of 16 bytes $(s_0^r, s_1^r, \cdots, s_{15}^r)$ and the key K^r to the round is split into 16 bytes $(k_0^r, k_1^r, \cdots, k_{15}^r)$. The next state S^{r+1} is the output of the r^{th} round. The input and round key to the first round S^1 is (P $\oplus$ K) and K^1 respectively.

$$\begin{aligned} S^{r+1} = \{ & T_0[s_0^r] \oplus T_1[s_5^r] \oplus T_2[s_{10}^r] \oplus T_3[s_{15}^r] \oplus \{k_0^r, k_1^r, k_2^r, k_3^r\} \\ & T_0[s_4^r] \oplus T_1[s_9^r] \oplus T_2[s_{14}^r] \oplus T_3[s_3^r] \oplus \{k_4^r, k_5^r, k_6^r, k_7^r\} \\ & T_0[s_8^r] \oplus T_1[s_{13}^r] \oplus T_2[s_2^r] \oplus T_3[s_7^r] \oplus \{k_8^r, k_9^r, k_{10}^r, k_{11}^r\} \\ & T_0[s_{12}^r] \oplus T_1[s_1^r] \oplus T_2[s_6^r] \oplus T_3[s_{11}^r] \oplus \{k_{12}^r, k_{13}^r, k_{14}^r, k_{11}^r\}\} \end{aligned}$$

2.2 Clefia Block Cipher

Clefia is a small lookup table based 128-bit block cipher [21]. It has a generalized Feistel structure. There are three key lengths of 128, 192 and 256 bits defined in the specification [22]. For the 128-bit key based specification the input is of 16 bytes, P_0 to P_{15}, grouped into 4 byte words. For each of the 18 rounds in the cipher, the first and third words are fed into functions F_0 and F_1 respectively. These functions are non-linear in nature. The collective outputs of F_0 and F_1, is known as F functions. These outputs are ex-ored with second and fourth words. In addition to this, at the beginning and end of the encryption the second and the fourth words are whitened.

To create the non-linearity in the F functions two sboxes S_0 and S_1 are used. These sboxes are in the form of 256 byte lookup tables, from each F function they are invoked twice. This makes a total of eight table lookups per round. Thus, for the entire encryption 144 such lookups are needed. Following are the equations of the functions F_0 and F_1:

$$F_0 : \{y_0, y_1, y_2, y_3\} = (S_0[x_0 \oplus k_0], S_1[x_1 \oplus k_1], S_0[x_2 \oplus k_2], S_1[x_3 \oplus k_3]) \cdot M_0$$
$$F_1 : \{y_0, y_1, y_2, y_3\} = (S_1[x_0 \oplus k_0], S_0[x_1 \oplus k_1], S_1[x_2 \oplus k_2], S_0[x_3 \oplus k_3]) \cdot M_1$$

Along with four round keys, k_0, k_1, k_2, k_3 the F functions take four input bytes x_0, x_1, x_2 and x_3. After the sbox lookups, the bytes are diffused by multiplying them with (4×4) matrices M_0 and M_1. Following are the structure of the matrices M_0 and M_1:

$$M_0 = \begin{pmatrix} 1 & 2 & 4 & 6 \\ 2 & 1 & 6 & 4 \\ 4 & 6 & 1 & 2 \\ 6 & 4 & 2 & 1 \end{pmatrix} \quad M_1 = \begin{pmatrix} 1 & 8 & 2 & A \\ 8 & 1 & A & 2 \\ 2 & A & 1 & 8 \\ A & 2 & 8 & 1 \end{pmatrix}$$

The whitening requires four whitening keys WK_0, WK_1, WK_2 and WK_3 along with thirty six round keys $RK_0, \cdots, RK_{35}$. A two step key expansion process is used. Firstly, from the secret key a 128 bit intermediate key L is generated, using a GFN function [14]. Then, the round keys and the whitening keys are generated from this. The structure of Clefia is such that the knowledge of any set of 4 round keys $(RK_{4m}, RK_{4m+1}, RK_{4m+2}, RK_{4m+3})$, where $m \ mod \ 2 = 0$, is sufficient to revert the key expansion process to obtain the secret key.

2.3 Time Obfuscating Countermeasures

The main principle for cache-timing attacks is the profiling and analysis of timing information returned by the Time-Stamp Counter (TSC). An adversary uses RDTSC instructions to access these TSC for granular timing information. A very common countermeasure to thwart attacks using timing channel is to provide the adversary a modified timing information instead of the real one. The obfuscation of RDTSC [11] can be done in two ways, first by introducing the concept of the real offset, which is the insertion of a real-time delay that stalls RDTSC execution, and then using the apparent offset that is by modification of the return value of the instruction by a small amount. The time is conceptually divided into *epochs* (denoted by E) to calculate these offsets. Epochs vary in length randomly from 2^{e-1} to $2^e - 1$ cycles, where e is denoted as the current level of obfuscation.

The real offset delays the execution of each RDTSC until a random time in the subsequent epoch, and this requires that the TSC register is always read on an epoch boundary. The current execution will be stalled until the end of the current epoch, Whenever an RDTSC instruction is encountered. TSC register will be read, on the epoch boundary. The instruction will be stalled continuously for a random number of cycles in the range $[0, E)$ of the subsequent epoch. The real offset denoted by D_R, is defined by the sum of these two stalls. These modifications result in hindering the malicious processes in user-space from making fine grain timing measurements to a granularity smaller than 2^{e-1}. This makes micro-architectural events undetectable as long as the largest difference between on-chip micro-architectural latencies is less than 2^{e-1}.

2.4 Hardware Performance Counters

Hardware Performance Counters (HPCs) are a set of special purpose registers, which are present in most of the modern microprocessor's Performance Monitoring Unit (PMU). These registers can be programmed to store the number of occurrences of different types of hardware and software events related to the execution of a program, such as cache misses, retired instructions, retired branch instructions, and so on. HPCs were primarily designed to debug the performance of complex software systems, but currently, they are widely used for collecting the run-time behavioral information of software execution. HPCs work along with the event selectors, which specify the hardware events to be monitored and a digital logic which increments a counter based on the occurrence of the specified hardware events. These performance counters can be accessed very fast without affecting or slowing down any software execution. Some of the recent literature [25–27] have used HPCs to dynamically profile a system.

The most useful mode of operation of PMUs is the interrupt-based mode. The main working principle behind this mode of operation is, a system interrupt is generated when a specified event occurs more than or equal to a predefined threshold value or a preset amount of time has elapsed. This mode of operation makes both event-based and time-based sampling possible. High-level libraries like PAPI [18], OProfile [15] provide interfaces to HPCs. Linux perf [17] among them is a widely used new implementation of performance counters support for all Linux 2.6+ based systems, which we can access from user-space. This tool is capable of providing per-process, per-CPU, and system-wide statistical profile. We used this tool for our experimentation purpose. Perf tool is based on Linux perf_event_open() system call, which can be used to profile system in very low granularity. Almost every popular operating systems have HPC-based profilers, though the type and number of hardware events may vary across different Instruction Set Architectures [9].

2.5 Metrics of Evaluation

Several formal security metrics [12] have been proposed in the literature to evaluate various attack methods and compare different cryptographic design with side channel perspective.

Success Rate. A side channel attack is defined as an experiment $Exp_{A_{E_K,L}}$, where $A_{E_K,L}$ is an adversary with time complexity τ, memory complexity m and making q queries to the target implementation of the cryptographic algorithm, where K denotes the key space and a leakage model for the key is denoted by L. In the experiment, for any k chosen randomly from K, when the adversary, $A_{E_k,L}$ outputs the guessing vector g, the attack is considered as a success if the corresponding key class denoted as $s = f(k)$ is such that $s \in g$. The success or failure of the attack is indicated by '0' or '1', returned by the experiment. The o^{th} order *success rate* of the side channel attack $A_{E_K,L}$ against the key classes is defined as [12]:

$$Succ^{o}_{A_{E_K},L}(\tau, m, k) = Pr[Exp_{A_{E_K,L}} = 1]$$

Guessing Entropy. The above-mentioned metric for an o^{th} order attack implies the success rate for an attack where the remaining workload is o-key classes. Thus the attacker has a maximum of o-key classes to which the required k may belong. While the success rate for a given order is fixed with respect to the remaining work load, the *guessing entropy* provides a more flexible definition for the remaining work load. It actually measures the average number of key candidates to test after the attack. The Guessing Entropy of the adversary $A_{E_k,L}$ is defined as [12]:

$$GE_{A_{E_K,L}}(\tau, m, k) = E[Exp_{A_{E_K,L}}]$$

In the next section we analyze a new source of side-channel in the form of `instruction` count using AES as a case study.

3 Information Leakage Due the Event Instruction Count

In this section, we describe the basics of performance monitoring tools which we used to observe the total number of retired instructions during the execution of the encryption process. In this context, we have explored the types of instructions using a more detailed analysis. The tools that we used are *perf* and *PAPI*, which we discuss next.

Perf is a performance analyzing tool in Linux which is available to all user level processes and has been included in the Linux kernel source tree for version 2.6.31 onwards. This user-space tool can be accessed from command line providing many sub-commands. It is capable of statistical profiling of the entire system by instrumenting the hardware performance counters. *perf* supports a list of hardware events to monitor, like cache-misses, branch-misses, cpu-cycles, etc.

For our proposed attack we observe the event *instruction* to analyze the source of side-channel. The total number of instructions executed for a single iteration of an encryption algorithm (say AES) is measured using the following command in perf:

```
perf stat − e instructions ./aes <plaintext>
```

The executable `./aes` has a specific secret key and provides an output ciphertext value for a given `<plaintext>`.

PAPI. One limitation of the perf tool is that we can observe the total number of instruction executed but can not further distinguish between types of instructions. **P**erformance **A**pplication **P**rogramming **I**nterface (PAPI) provides a user with a consistent interface and methodology for monitoring performance counters and can even show counts of finer hardware events like number of control instructions, number of data transfer instructions, etc.

PAPI is more sophisticated than perf tool since it provides a larger number of hardware events for monitoring. The total monitored instruction count, as measured by perf tool, is further divided into specialized hardware events in PAPI interface such as:

– `PAPI_BR_INS`: This event can be used to measure the total branch instructions executed for an encryption algorithm.
– `PAPI_LST_INS`: This event can be used to measure the total load/store instructions executed for an encryption algorithm.

The above two events provide us with the handle to analyze and investigate the source of information leakage.

3.1 Correlation of Cache Events to Instruction Counts

Efficient implementations of block ciphers use lookup tables to perform the computations involved in encryption and decryption operations. As described in Sect. 2.1, the look up table accesses during the encryption process are dependent on both the input plaintext and the secret key. The respective memory addresses of these lookup table accesses vary depending on the input plaintext as well as the secret key. The cache timing attacks reported in literature exploit the non-uniformity in access times of these table lookup requests to retrieve the secret information. The non-uniformity of timing observations are typically attributed to cache memory events such as cache hit and miss.

A cache-miss occurs whenever the requested data is not present in the cache. On a cache-miss event, the memory controller needs to fetch the requested data from the main memory and loads it into the cache. Thus on a cache miss, a memory element from the cache memory (which is being replaced by the newly requested data block) is written back in the main memory, followed by loading of a new data element in the particular location of the cache. The decision of which block to be replaced for a new request is partially determined by the virtual to physical address mapping of the data block and partially governed by the cache replacement strategies implemented by the memory controller. Thus the number of instructions executed by the processor has a direct correspondence to the event encountered by the cache memory. Since on a cache miss, the processor requests the memory element to be brought from main memory to the cache; this event is inherently performed with a higher number of instructions. However, in the case of a cache-hit event, the processor will not issue any additional instructions to load the data from the memory as the required data is already present in the cache. This brings us to the conclusion that the cache events will have an alternative effect on the instruction count event.

3.2 Profiling the Instruction Counts

In the previous subsection, we have elaborated that individual cache events have a direct correspondence to the instruction counts. In this subsection, we demonstrate that the average deviation for instruction counts have a similar profile as that of the timing profile constructed for cache timing attacks. We conduct an experiment on the OpenSSL [23] AES encryption. The plaintext byte p_0 is varied randomly from 0 to 255, keeping all other bytes unchanged. Initially, we obtained a timing profile using RDTSC instructions as shown in Fig. 1a for the key byte k_0. Keeping the experimental setup unchanged, next, we observed the instruction profile as in Fig. 1b which plots the deviation from the mean of the total instruction count. The deviation from the average value of the monitored event for each byte of p_0 is shown in Fig. 1. This graph is known to be the characteristics curves for the monitored events. A significant deviation from the average for a particular key byte shows the existence of information leakage. We can easily see that both the characteristics curves for timing and instructions are similar and hence supports our claim that instruction can be used as an alternative to the timing profile.

However, the event cache-misses can also be observed from the HPCs, and thus we had replicated the same experiment replacing instruction count with cache misses, and the cache miss profile is illustrated in Fig. 1c. The event

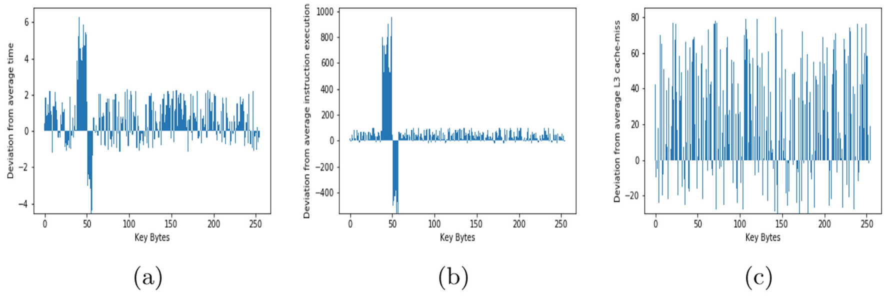


Fig. 1. Deviation of total execution time (a), total instruction counts (b), and total L3 cache-misses (c) during an AES encryption operation from average for different plaintext byte p_0 generated randomly keeping the other bytes unchanged

monitored through PAPI observes the cache misses from all the three levels of cache. This information is highly noisy as in the figure and bears no resemblance to the timing observation. The reason for this behavior is mainly because the cache misses plotted in the figure are for the L3 cache, while we assume the lookup accesses for encryptions are mostly happening from L1 and L2 caches. Liu et al. presented the practicality of a cross-core, cross-VM LLC-cache based *Prime+Probe* attack [10] and showed that a proper eviction algorithm is needed to mount a successful attack, which is difficult in the real noisy environment.

We can conclude from the figures presented in Fig. 1 that instruction count bears a direct correspondence to the timing side channel rather than the event cache misses. In the next section, we will move a step deeper to validate the claims that we have made in this section.

3.3 Analyzing Load/Store Instruction Counts

In the previous subsection, we have observed that the total instruction count generates similar profiles as that of timing attacks. In this section, we further explore the different types of instructions using the tool PAPI to investigate the type of instructions responsible for generating similar profile as timing. The hardware events which are related to the total instructions are given as follows:

1. Data Manipulation Instructions
   - Consists of arithmetic instructions, logical instructions, shift instructions, etc. Provides computational capabilities to the computer by performing different operations on data.
   - The hardware events for monitoring these instructions are `PAPI_INT_INS`, `PAPI_FP_INS`, etc., which measure the total integer instructions, total floating point instructions respectively, spawned by the processor. However, the PAPI tool does not provide the handle to observe these hardware events.
2. Data Transfer Instructions
   - Transfer data between memory and registers, register & input or output, and between processes register without changing the data content.
   - PAPI tool gives the handle to monitor these instructions using the hardware event `PAPI_LST_INS`.
3. Program Control Instructions
   - Direct or change the flow of a program. Mainly consists of all the branch instructions.
   - We can observe these instructions with PAPI tool by using the hardware event `PAPI_BR_INS`.

We demonstrate another experiment as the previous one to find and validate the actual type of instructions which are mainly responsible for generating the profile same as timing. Here, we observe the events `PAPI_BR_INS` (type 3) and `PAPI_LST_INS` (type 2) using the PAPI tool to get the characteristics curve for each event. The cache-miss and cache-hits are related to data transfer instructions as a cache miss will result in loading data into a particular cache line.

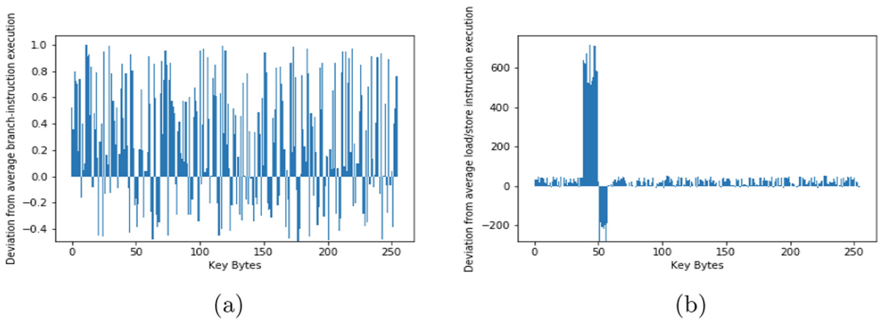


Fig. 2. Deviation of total branch instructions (a) executed and total load/store instructions (b) executed during an AES encryption operation from average for different plaintext byte p_0 generated randomly keeping the other bytes unchanged

We can validate this from Fig. 2b, where we can observe, the profile generated by the event `PAPI_LST_INS` has the same resemblance to the profile of the total instructions. However, Fig. 2a, shows that the profile generated by the branch instructions are not at all similar to that of the total instructions, and hence works as noise in the observations. Figure 2 shows both the characteristic curves for branch instructions and load/store instructions respectively and the behavior of the characteristics plots validates that load and store instructions bear a direct resemblance to the timing characteristics and thus are a potential source of leakages.

In the next section, we present a formal description of the proposed attack methodology, IPA, with a case study on AES block cipher.

4 Instruction Profiling Attack Description

The previous section describes the hardware event `instruction` as a potential side-channel threat because of the resemblance of its profile to that of time. In this section, we give a formal description of our attack methodology that we have used in this paper.

4.1 Instruction Count Analysis for AES

For an AES encryption each table T_0, T_1, T_2, and T_3 is accessed four times in every round for the first nine rounds, while table T_4 is accessed 16 times in the final round. In all, there are 160 table accesses. Following the analysis presented in [19], if n_h is the number of cache hits and n_m is the number of misses, then the total instructions executed during the encryption process can be written as:

$$\begin{aligned} I &= n_h * I_h + n_m * I_m \\ &= n_h * I_h + (160 - n_h) * I_m \end{aligned}$$

Here, I_h is the number of instructions executed when a cache-hit occurs and I_m is the same when a cache-miss occurs. Note, we are focussing on data transfer instructions, as they are the suspected contributors to the leakage. Furthermore, it may be emphasized here that when we perform the actual attacks, we consider the variation of the total instruction count, and not exclusively instructions of any specific type. This improves the practicality of the attack. The difference of instruction counts between two encryption processes can be written as:

$$\begin{aligned} \Delta I &= \Delta n_h * \Delta I_h + \Delta n_m * \Delta I_m \\ &= \Delta n_h * (\Delta I_h - \Delta I_m) \end{aligned}$$

Now, the parameters $\Delta n_h, \Delta I_h$, and ΔI_m depend on the cache-access patterns. The difference in number of hits, Δn_h, occurs because of the differences in accessing the Tables T_0 to T_3 during the encryption for AES. The difference $(\Delta I_h - \Delta I_m)$ depends on both plaintexts and the key, which are inputs to the cipher. So, by monitoring and statistically analyzing the instruction count we can obtain a profile which is dependent on the secret key, and thus potentially determine it by comparing with templates for known keys.

4.2 Description of IPA

In this subsection, we describe the proposed attack methodology taking Bernstein's attack on AES as a case study [13]. The proposed attack consists of three different phases: offline profiling, online attack, and correlation, which are discussed below.

Offline Profiling. In the profiling phase, we generate a set of random plaintext $P = \{p_0, p_1, \cdots, p_l\}$ and submit each of them to the encryption server with a known secret key k. We observe the total instruction count for each encryption, which can be written as $ins(E_{AES}(p_i, k))$. We store the average instruction count for each byte and for each value of that byte in a matrix $I[16][256]$. This can be formally stated as, for each plaintext p_i $(0 \leq i \leq l)$ and for each byte $(0 \leq j \leq 15)$ we successively compute the elements of the matrix $I[16][256]$ as below:

$$I[j][p_{j,i}] = I[j][p_{j,i}] + ins(E_{AES}(p_i, k))$$

where, $p_{j,i}$ is the j^{th} byte of the i^{th} plaintext. Eventually, we have for $0 \leq j \leq 16$ and $x \in \{0, 1, \cdots, 255\}$

$$I[j][x] = \sum_{\{i | p_{j,i} = x\}} ins(E_{AES}(p_i, k))$$

We then calculate the average number of instructions taken by each byte for each value of that byte using

$$v[j][y] = \frac{I[j][y]}{num[j][y]} - \frac{\sum_j \sum_x I[j][x]}{\sum_j \sum_x num[j][x]}$$

where, $num[16][256]$ stores the total number of measurements per value of a byte value. This phase profiles the encryption server for randomly chosen but known plaintexts and a known key.

Online Attack. In this phase, we again generate a random set of plaintexts $P' = \{p'_0, p'_1, \cdots, p'_l\}$. We follow the same approach as discussed in profiling phase and calculate two matrices I' and num' for the unknown key k'. We then calculate the matrix v' as defined before.

Correlation. In this phase, we correlate the two matrices v and v' obtained from the previous steps. A matrix $c[16][256]$ is created using the following definition for $0 \leq j \leq 15$ and $0 \leq u \leq 255$.

$$c[j][u] = \sum_{w=0}^{255} v[j][w] \cdot v'[j][u \oplus w]$$

The elements of the matrix c are then sorted in decreasing order for each row. The highest correlated key value for a particular byte is the candidate key for that key byte.

The results of the correlation phase will provide us with partial secret key recovery based on the quality of instruction profile for the known key. The full secret key can be recovered by *Brute Force* search for the remaining secret key bytes with very narrow search space.

In the following section, we discuss and validate our claims using experimental results in two different environments, and for two different block ciphers.

5 Results and Discussion

In this section, we focus on the performance and qualitative evaluation of the proposed attack scheme. We demonstrate the instruction profiling attack on two very well-known block ciphers such as AES and Clefia, implementation provided by OpenSSL library. To illustrate the performance of this attack we follow the same principle demonstrated in the work by Bernstein on AES [4] and Rebeiro et al. on Clefia [21]. The attack has been performed in following steps:

- `Instruction Profiling for Known key`: In this phase of the attack, the adversary observes the total instruction counts from the HPCs for an execution of AES/Clefia. Following the steps discussed in the previous section, we construct an instruction profiling table consisting of the cumulative values of the instruction counts suffered by the executable under the assumption of a known key.
- `Instruction Profiling for Unknown key`: In this phase, similar profile for instruction counts are constructed for the same executable, but with the assumption that the key is unknown.

Table 1. Description of experimental setups

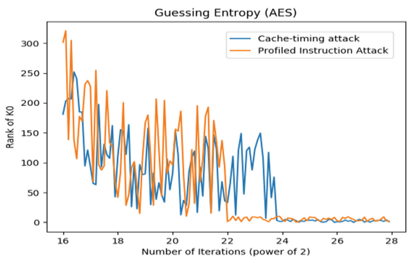
1	Setup 1	Intel Core i5-4570 CPU with 3.20 GHz clock frequency, 256 KB of L1 Data Cache, 1 MB L2 Data Cache, 6 MB L3 Data Cache
2	Setup 2	AMD A10-8700P CPU with 1.8 GHz clock frequency, 320 KB of L1 Data Cache, 2 MB of L2 Data Cache

– `Correlating known and unknown key profiles`: In this phase, we chose very popular `Pearson's` correlation metric to determine the shift between the instruction profiles observed over the known and the unknown keys.

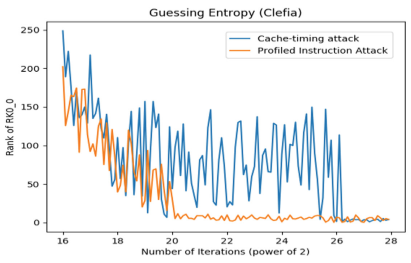
It has to be noted that the entire analysis is performed based on a particular byte of the input. So to retrieve the entire key, this analysis has been carried out for each byte of the input. The effect of each byte being independent of each other, this analysis is usually performed concurrently on each byte using a divide and conquer approach which adds to the elegance of the attack and does not add up to the time complexity. In this paper, without loss of generality, we have demonstrated all our experiments on the first secret byte of the key. Also, we have validated all the experiments in two different environments like Intel and AMD systems to get a generalized performance measure of the proposed technique. Table 1 briefly states different setups used in our experiment.

5.1 Performance Evaluation of IPA in Comparision to Classical Timing Attack

In this subsection, we evaluate the efficiency of the proposed attack methodology to the timing attack proposed in [4] using the formalism for Guessing Entropy as discussed in Sect. 2.5. The convergence of the plot to a lower rank for a particular key byte signifies the successful retrieval of that byte. Figure 3a illustrates the



(a)



(b)

Fig. 3. Guessing Entropy plots of proposed attack in comparison to cache-timing attack for setup 1 to predict secret key-byte k_0 of AES (a) and round key-byte $RK0\_0$ of Clefia (b) respectively (Color figure online)

guessing entropy plots of cache-timing attack and profiled instruction attack to predict the secret key byte k_0 of AES on setup 1 (as in Table 1).

We illustrate in Fig. 3a that cache-timing attack needs 2^{24} iterations, whereas, IPA requires 2^{22} iterations of the AES encryption algorithm to correctly predict the secret key byte k_0. Thus IPA needs to profile instruction counts from perf for orders of magnitude lesser than cache-timing attack to correctly retrieve the secret key bytes. Similarly, Fig. 3b shows that, the proposed attack methodology is much faster to predict the round key byte $RK0\_0$ of Clefia encryption by profiling 2^{20} iterations in comparison to 2^{26} iterations of cache-timing attack.

5.2 Performance of Timing Attack in Presence of Time Obfuscation

A popular countermeasure to obfuscate timing channel is to randomize the timing observation from the RDTSC instruction. One such implementation is proposed in [11] named Timewarp. In this paper, we show that the classical timing attack fails to retrieve the correct key bits when timewarp has been implemented. The *blue* lines in Fig. 4 displays the ineffectiveness of the cache-timing attack in the presence of timewarp defense mechanism. The *blue* lines plotted in Fig. 4a and b shows that even after 2^{28} iterations of both the AES and Clefia algorithms, the cache timing attack fails to predict the correct secret key bytes.

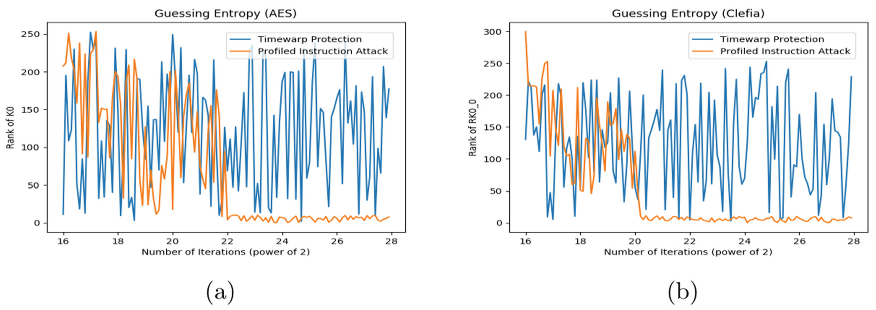


Fig. 4. Guessing Entropy plot of proposed attack to predict key-byte k_0 of AES (a) and round key-byte $RK0\_0$ of Clefia (b) in comparison to cache-timing attack in presence of timewarp implementation on setup 1 (Color figure online)

5.3 Performance of IPA in Presence of Timewarp

In this subsection, we perform the same attack strategy using instruction count from HPCs in the presence of the time obfuscation countermeasure: Timewarp, implemented in the system. We elaborate the results with Fig. 4, which shows the competency of our proposed attack method to retrieve the secret key bytes of the encryption algorithm even in the presence of timewarp defense mechanism. The figure demonstrates the results for both AES and Clefia algorithms.

The *yellow* lines in Fig. 4a and b show that the proposed profiled instruction attack can retrieve the secret key bytes even in the presence of timewarp implementation with 2^{20} and 2^{22} iterations of AES and Clefia algorithms respectively. The line plotted with *yellow* in Fig. 4 establishes the superiority of the proposed IPA to cache-timing attack as shown in *blue* plotted line, since it fails to retrieve the secret key-byte k_0 of AES and round key-byte $RK0\_0$ of Clefia respectively in the presence of the defense mechanism.

Retrieving the Secret Keys. Here, we discuss the effectiveness of our proposed attack methodology regarding the extraction of the secret key. We conducted each of the experiments in the presence of timewarp implementation. Here, we present the results for both AES and Clefia and for both the setups.

AES. In this section, we demonstrate the full-key recovery of the AES-128 implementation. As explained in the earlier sections, the entire key recovery can be done in a divide and conquer approach such that, all the key bits can be retrieved simultaneously since there is no mutual dependence. We have performed our experiments over several random key sequences of 128 bits.

Without loss of generality, we illustrate all of our results on a randomly chosen bit sequence. The **bold** bytes in the following results represent the correctly predicted secret key bytes.

Table 2 presents the final retrieved key bytes of AES using the proposed IPA in the presence of timewarp implementation. We can clearly see from Table 2 that the proposed IPA is able to retrieve the correct secret key bytes apart from k_2, k_8 and k_{11} in Setup 1 and k_5 and k_{11} in Setup 2 respectively, which we could later recover with brute-force search with lesser search space. This result shows the high effectiveness of the proposed attack methodology in recovering the secret keys of AES even in the presence of time obfuscation defense.

Table 3 shows the retrieved key bytes (having highest correlation value) of AES for both the setups using the cache-timing attack. We can clearly observe

Table 2. Correctly retrieving secret key of AES using IPA with Timewarp countermeasure

		k_0	k_1	k_2	k_3	k_4	k_5	k_6	k_7	k_8	k_9	k_{10}	k_{11}	k_{12}	k_{13}	k_{14}	k_{15}
	Actual Key	e2	6d	b3	f5	64	42	c6	92	3e	0f	96	b6	a1	52	9d	86
Predicted Key	Setup 1	**e2**	**6d**	a4	**f5**	**64**	**42**	**c6**	**92**	3a	**0f**	**96**	ae	**a1**	**52**	**9d**	**86**
	Setup 2	**e2**	**6d**	**b3**	**f5**	**64**	b8	**c6**	**92**	**3e**	**0f**	**96**	34	**a1**	**52**	**9d**	**86**

Table 3. Fail to retrieve secret key of AES using timing channel with Timewarp countermeasure

		k_0	k_1	k_2	k_3	k_4	k_5	k_6	k_7	k_8	k_9	k_{10}	k_{11}	k_{12}	k_{13}	k_{14}	k_{15}
	Actual Key	e2	6d	b3	f5	64	42	c6	92	3e	0f	96	b6	a1	52	9d	86
Predicted Key	Setup 1	d9	ea	b3	24	4d	08	6d	64	4f	fc	c2	6d	af	dc	64	88
	Setup 2	72	8a	3b	6e	7f	4c	dc	1c	7f	42	af	a6	76	20	7c	b2

Table 4. Fail to retrieve secret key of AES using branch instructions with Timewarp countermeasure

		k_0	k_1	k_2	k_3	k_4	k_5	k_6	k_7	k_8	k_9	k_{10}	k_{11}	k_{12}	k_{13}	k_{14}	k_{15}
	Actual Key	e2	6d	b3	f5	64	42	c6	92	3e	0f	96	b6	a1	52	9d	86
Predicted Key	Setup 1	57	48	2f	6e	66	d2	16	73	23	82	bc	02	04	01	4f	96
	Setup 2	4f	96	64	82	c4	56	a5	8c	16	96	3c	d3	b0	10	cb	dc

from the table that, the cache-timing attack fails to retrieve any of the secret key bytes of AES for both the setups (as expected). Table 4 shows the final retrieved values of the AES secret keys considering branch instructions for profiling. We can easily verify from the table that, for profiling using branch instructions the attack method can not retrieve any of the secret key bytes correctly, which validates our claim that branch instructions do not play any role for the secret information leakage for these class of ciphers. This is also intuitive as block cipher implementations do not have conditional branches, which could leak information about the key. But the interesting part is that there the overall instruction count can still be exploited, because of the reasons as mentioned above to determine the secret key.

Clefia. In this subsection, we demonstrate the full-key recovery of Clefia implementation. We have experimented using different secret keys for Clefia to validate our proposed attack methodology, though for the demonstration purpose the 128-bit secret Clefia key that we considered is 6a 1a 58 e2 12 30 35 e7 fd aa 3b 6e f4 8e d4 5f. The Round Keys corresponding to the given secret key are given in Table 5. Without loss of generality, we show the recovery of round key RK0_0. We perform the experimentation with the assumption of clean cache at the start of every encryption. The correct value of the round keys depend on the previous round keys for Clefia; thus we considered at least 2^{20} iterations so that the correlation value of the predicted key is at least twice the higher than all other probable keys. Like previous results, the bytes written in **bold** face represent the correct key bytes.

Tables 6 and 7 presents the final retrieved values for the $RK0$ round key in Setup 1 and Setup 2 respectively using proposed IPA. Both the table shows the top four candidate for the probable round key bytes. `The values in the`

Table 5. Round keys for Clefia

RK0	0xbe	0xf8	0xe7	0xae
RK1	0x75	0x61	0xb8	0x30
RK2WK0	0x91	0xe1	0x3e	0x46
RK3WK1	0x34	0x1f	0x5f	0x6f
RK4	0x70	0xc7	0xcc	0xd8
RK5	0xb8	0x90	0xb3	0xec

Table 6. Retrieving round keys RK0 for Clefia in Setup 1 using total instruction count with Timewarp countermeasure

	Top-4 probable RK0 round keys (correlation value)			
RK0_0	**0xbe (250.167)**	0x6b (30.158)	0x7e (31.148)	0xee (23.137)
RK0_1	**0xf8 (260.326)**	0xd1 (34.242)	0xfc (33.682)	0xe8 (31.024)
RK0_2	**0xe7 (255.388)**	0x9f (57.648)	0x31 (56.957)	0x7a (54.515)
RK0_3	**0xae (255.851)**	0x87 (33.130)	0xbe (32.455)	0x41 (30.312)

Table 7. Retrieving round keys RK0 for Clefia in Setup 2 using total instruction count with Timewarp countermeasure

	Top-4 probable RK0 round keys (correlation value)			
RK0_0	**0xbe (260.166)**	0x6b (34.153)	0x7e (31.147)	0xee (30.130)
RK0_1	**0xf8 (265.456)**	0xd1 (33.478)	0xfc (32.147)	0xe8 (31.200)
RK0_2	**0xe7 (247.457)**	0xae (57.124)	0x43 (57.008)	0x95 (54.214)
RK0_3	**0xae (259.567)**	0x87 (47.247)	0xbe (46.211)	0x41 (40.589)

Table 8. Retrieving round keys RK0 for Clefia in Setup 1 using timing attack with Timewarp countermeasure

	Top-4 probable RK0 round keys (correlation value)			
RK0_0	0xce (314.699)	0x65 (289.491)	0x20 (276.232)	0xae (213.873)
RK0_1	0xac (775.449)	0x68 (761.411)	0xbb (603.751)	0xb2 (577.428)
RK0_2	0x56 (453.751)	0xd7 (417.697)	0xc8 (347.645)	0xfe (249.147)
RK0_3	0x37 (598.248)	0xac (548.479)	0x6b (497.268)	0xd5 (457.314)

Table 9. Retrieving round keys RK0 for Clefia in Setup 2 using timing attack with Timewarp countermeasure

	Top-4 probable RK0 round keys (correlation value)			
RK0_0	0xd7 (478.324)	0xba (421.984)	0x2e (394.157)	0xcf (350.496)
RK0_1	0x1e (367.459)	0xf9 (314.496)	0xa1 (296.549)	0xd9 (247.693)
RK0_2	0x8a (724.967)	0x4d (695.349)	0xa0 (645.945)	0x09 (634.235)
RK0_3	0xe4 (676.935)	0x45 (645.453)	0x00 (601.239)	0xda (509.486)

`braces are the correlation value of the probable keys`. We can easily observe from the tables that the proposed IPA is able to recover all the bytes of the round key $RK0$ for both the setups in the presence of timewarp defense mechanism. The correlation values for the correctly predicted key bytes are much greater than the subsequent candidate keys.

Tables 8 and 9 presents the final retrieved values for the $RK0$ round key in Setup 1 and Setup 2 respectively using cache-timing attack like the previous

Table 10. Retrieving round keys RK0 for Clefia in Setup 1 using branch instructions with Timewarp countermeasure

	Top-4 probable RK0 round keys (correlation value)			
RK0_0	0x2f (869.143)	0xda (649.786)	0x38 (575.880)	0x62 (561.020)
RK0_1	0xde (668.557)	0x3e (615.218)	0xdd (587.463)	0xab (499.769)
RK0_2	0xdd (584.990)	0x6d (512.642)	0xd3 (456.590)	0xaa (448.524)
RK0_3	0xd (703.281)	0x90 (619.583)	0x3f (577.043)	0x3e (552.067)

Table 11. Retrieving round keys RK0 for Clefia in Setup 2 using branch instructions with Timewarp countermeasure

	Top-4 probable RK0 round keys (correlation value)			
RK0_0	0x2f (749.457)	0xda (657.457)	0x38 (602.983)	0x62 (597.237)
RK0_1	0xde (457.698)	0x3e (421.697)	0xdd (403.743)	0xab (347.573)
RK0_2	0xdd (726.147)	0x6d (689.478)	0xd3 (623.951)	0xaa (599.974)
RK0_3	0xd (714.649)	0x90 (687.967)	0x3f (567.697)	0x3e (547.546)

Table 12. Top-2 predicted round keys of Clefia

Round key bytes	Correct key	Predicted key (Setup 1)	Predicted key (Setup 2)
RK0_0	be	**be** (250.548) 6b (33.518)	**be** (331.478) 72 (30.347)
RK0_1	f8	**f8** (236.478) d1 (34.398)	**f8** (347.149) 0e (32.478)
RK0_2	e7	**e7** (259.496) 9f (31.759)	**e7** (312.479) e9 (35.478)
RK0_3	ae	**ae** (249.247) 87 (30.974)	**ae** (299.647) 5e (31.479)
RK1_0	75	**75** (239.479) 6e (29.647)	**75** (357.457) 14 (29.475)
RK1_1	61	**61** (213.795) 0a (37.198)	**61** (378.147) 2d (47.149)
RK1_2	b8	**b8** (297.347) 54 (30.789)	**b8** (249.647) 64 (36.759)
RK1_3	30	**30** (267.126) 7c (31.496)	**30** (432.148) 1f (26.478)
RK2_0 + WK0_0	91	**91** (257.214) d3 (34.189)	**91** (496.487) 77 (31.478)
RK2_1 + WK0_1	e1	**e1** (269.147) b6 (33.698)	**e1** (249.657) 36 (32.768)
RK2_2 + WK0_2	3e	**3e** (259.347) fb (31.478)	**3e** (387.162) 32 (26.479)
RK2_3 + WK0_3	46	**46** (249.347) df (29.678)	**46** (321.338) 3c (47.147)
RK3_0 + WK1_0	34	**34** (298.147) 87 (35.148)	**34** (410.814) cc (43.549)
RK3_1 + WK1_1	1f	**1f** (267.348) 8d (31.987)	**1f** (490.703) c6 (29.647)
RK3_2 + WK1_2	5f	**5f** (249.347) ff (34.158)	**5f** (228.757) f9 (33.679)
RK3_3 + WK1_3	6f	**6f** (219.347) 38 (36.489)	**6f** (353.479) f2 (29.624)
RK4_0	70	**70** (278.498) 1a (32.489)	**70** (228.749) 7e (45.697)
RK4_1	c7	**c7** (264.369) b0 (29.634)	**c7** (249.647) 53 (49.547)
RK4_2	cc	**cc** (249.149) 66 (34.214)	**cc** (349.248) 04 (23.452)
RK4_3	d8	**d8** (278.694) 24 (28.365)	**d8** (324.479) 2e (32.479)
RK5_0	b8	**b8** (324.496) 68 (49.324)	**b8** (367.457) 33 (36.139)
RK5_1	90	**90** (257.354) 83 (31.647)	**90** (246.479) e6 (29.498)
RK5_2	b3	**b3** (264.236) 2c (26.498)	**b3** (226.714) bf (36.249)
RK5_3	ec	**ec** (321.698) 43 (45.268)	**ec** (314.789) f2 (33.149)

results. We can observe from the tables that cache-timing attack fails to recover the round key $RK0$ for both the setups in the presence of timewarp defense mechanism. The correlation values for the top four candidate keys are very close to each other, in both the setups, thereby creating the difficulty in predicting the actual secret key correctly. Similarly, Tables 10 and 11 presents the final retrieved values for the $RK0$ round key in Setup 1 and Setup 2 respectively by profiling through branch instructions, which shows the expected inefficiency in retrieving the secret keys using branch instructions.

Table 12 shows the retrieval of all the Clefia round keys using IPA in the presence of timewarp defense mechanism to show the efficiency of the proposed attack. The table shows the `top two candidate key` with respective correlation in the braces for both the setups. We observe that full recovery of the secret key is possible with the proposed IPA for Clefia encryption.

5.4 Success Rate of the Proposed IPA

The success rate for a side-channel attack is represented as the fraction of the secret key bytes recovered. For any successful side-channel attack, success rate increases with the number of iterations of the monitored encryption algorithm. Here, we present the success rate of the proposed IPA with the cache timing attack in the absence of timewarp defense mechanism to show the effectiveness of the IPA in retrieving the secret key bytes with lesser time. Figure 5 shows the success rate of both IPA and cache-timing attack in Setup 1. The figures represent the part of the secret keys retrieved successfully with the increase in the number of iterations. Figure 5a and b show the success rate in retrieving the secret key bytes of AES and Clefia. The lines plotted with *blue* color represent the success rate of the proposed IPA, and the success rate of classical cache-timing attacks is represented by *yellow* colored lines. It is to be noted that the *yellow* line is always below the *blue* line for both AES and Clefia, signifying the better success rate of IPA than the classical timing attacks for both the ciphers.

In the next section, we discuss the practicality of the proposed attack methodology in different environments along with a possible countermeasure.

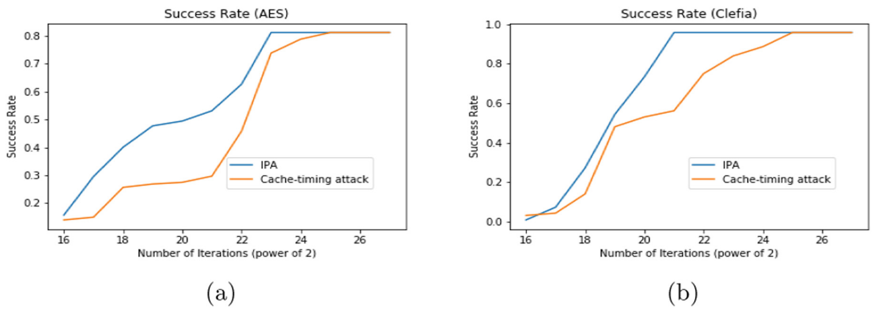


Fig. 5. Comparison of Success rate of IPA with cache-timing attack in Setup 1 in predicting secret key-bytes of AES (a) and Clefia (b) respectively (Color figure online)

6 Practicality of the Proposed Attack

The proposed attack methodology discussed in this paper is a primitive implementation, which does not require the assumption of shared cache memory between different users. This gives the attack an advantage over other types of attacks like Prime + Probe attacks. There are inherent protections in most of the modern processor architectures, like Intel SGX, which guard against cache trace based attacks using secure enclaves. However, the event instruction count reflects the effect of cache access patterns in spite of this security. We aim to explore these architectures in our future study.

A possible countermeasure for this attack is the implementation of block ciphers without requiring any table lookups. There are some modern cryptographic libraries like NaCl [5], which provide us the block cipher implementations having no table accesses. This provides an interesting approach to mitigate the proposed attack methodology.

7 Conclusion

In this paper, we have investigated a new side-channel leakage through the HPC event *instructions* and successfully designed an attack methodology, namely Instruction Profiling Attack (IPA). We demonstrate that surprisingly even total instruction counts can be utilized to perform side channel analysis on block ciphers, which because of the absence of conditional branch instructions were not targeted previously for instruction based attacks. In fact, we demonstrate that the proposed IPA has better performance than classical cache- timing attacks, and are better side channels than the customary timing information. We also attempt to bring out an implication of the attack, that defenses against timing attacks which are based on time fuzzing, will not be able to prevent the proposed IPA. We validate our claims with results for two different environments, namely Intel and AMD. The paper proves once again that the cache memory is an important artifact for side channel leakage; rather than trying to obfuscate channels like timing it is more important to design micro- architectures with security-awareness in the early phase of the design cycle.

References

1. Aciiçmez, O.: Yet another microarchitectural attack: : exploiting I-cache. In: Proceedings of the 2007 ACM Workshop on Computer Security Architecture, CSAW 2007, Fairfax, VA, USA, 2 November 2007, pp. 11–18 (2007)
2. Acıiçmez, O., Schindler, W., Koç, Ç.K.: Cache based remote timing attack on the AES. In: Abe, M. (ed.) CT-RSA 2007. LNCS, vol. 4377, pp. 271–286. Springer, Heidelberg (2006). https://doi.org/10.1007/11967668_18
3. Barreto, P.S.L.M.: The AES block cipher in C++
4. Bernstein, D.J.: Cache-timing attacks on AES. Techical report (2005)

5. Bernstein, D.J., Lange, T., Schwabe, P.: The security impact of a new cryptographic library. In: Hevia, A., Neven, G. (eds.) LATINCRYPT 2012. LNCS, vol. 7533, pp. 159–176. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33481-8_9
6. Bhattacharya, S., Rebeiro, C., Mukhopadhyay, D.: Unraveling timewarp: what all the fuzz is about? In: HASP 2013, The Second Workshop on Hardware and Architectural Support for Security and Privacy, Tel-Aviv, Israel, 23–24 June 2013, p. 8 (2013)
7. Federal Information Processing Standards Publication 197. Announcing the Advanced Encryption Standard (AES)
8. Granger, R., Page, D., Stam, M.: Hardware and software normal basis arithmetic for pairing-based cryptography in characteristic three. IEEE Trans. Comput. **54**(7), 852–860 (2005)
9. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B, 3C & 3D): System Programming Guide (2010)
10. Liu, F., Yarom, Y., Ge, Q., Heiser, G., Lee, R.B.: Last-level cache side-channel attacks are practical. In: 2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, 17–21 May 2015, pp. 605–622 (2015)
11. Martin, R., Demme, J., Sethumadhavan, S.: Timewarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In: 39th International Symposium on Computer Architecture (ISCA 2012), Portland, OR, USA, 9–13 June 2012, pp. 118–129 (2012)
12. Mukhopadhyay, D., Chakraborty, R.S.: Hardware Security: Design, Threats, and Safeguards, 1st edn. Chapman & Hall/CRC, Boca Raton (2014)
13. Neve, M., Seifert, J., Wang, Z.: A refined look at Bernstein's AES side-channel analysis. In: Proceedings of the 2006 ACM Symposium on Information, Computer and Communications Security, ASIACCS 2006, Taipei, Taiwan, 21–24 March 2006, p. 369 (2006)
14. Nyberg, K.: Generalized Feistel networks. In: Kim, K., Matsumoto, T. (eds.) ASIACRYPT 1996. LNCS, vol. 1163, pp. 91–104. Springer, Heidelberg (1996). https://doi.org/10.1007/BFb0034838
15. OProfile (2015). http://oprofile.sourceforge.net/news/
16. Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: the case of AES. In: Pointcheval, D. (ed.) CT-RSA 2006. LNCS, vol. 3860, pp. 1–20. Springer, Heidelberg (2006). https://doi.org/10.1007/11605805_1
17. perf: Linux profiling with performance counters (2015)
18. Performance Application Programming Interface (2016)
19. Rebeiro, C., Mondal, M., Mukhopadhyay, D.: Pinpointing cache timing attacks on AES. In: VLSI Design 2010: 23rd International Conference on VLSI Design, 9th International Conference on Embedded Systems, Bangalore, India, 3–7 January 2010, pp. 306–311 (2010)
20. Rebeiro, C., Mukhopadhyay, D., Bhattacharya, S.: Timing Channels in Cryptography: A Micro-Architectural Perspective. Springer Publishing Company, Incorporated, Cham (2014). https://doi.org/10.1007/978-3-319-12370-7
21. Rebeiro, C., Mukhopadhyay, D., Takahashi, J., Fukunaga, T.: Cache timing attacks on clefia. In: Roy, B., Sendrier, N. (eds.) INDOCRYPT 2009. LNCS, vol. 5922, pp. 104–118. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-10628-6_7
22. Sony Corporation: The 128-bit blockcipher Clefia: Algorithm specification (2007)
23. The OpenSSL Project. http://www.openssl.org

24. Unix Stack Exchange. https://unix.stackexchange.com/questions/2126/why-is-there-a-big-delay-after-entering-a-wrong-password
25. Wang, X., Karri, R.: Numchecker: detecting kernel control-flow modifying rootkits by using hardware performance counters. In: The 50th Annual Design Automation Conference 2013, DAC 2013, Austin, TX, USA, 29 May–07 June 2013, pp. 79:1–79:7 (2013)
26. Wang, X., Karri, R.: Reusing hardware performance counters to detect and identify kernel control-flow modifying rootkits. IEEE Trans. CAD Integr. Circuits Syst. **35**(3), 485–498 (2016)
27. Wang, X., Konstantinou, C., Maniatakos, M., Karri, R.: Confirm: detecting firmware modifications in embedded systems using hardware performance counters. In: Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2015, Austin, TX, USA, 2–6 November 2015, pp. 544–551 (2015)